Recursion

Lars-Henrik Eriksson

Functional Programming 1

Based on a presentation by Tjark Weber and notes by Sven-Olof Nyström
Comparison: Imperative/Functional Programming
Imperative programming perhaps suggests itself: machine code is imperative; hardware contains memory whose state can change. A corresponding theoretical model is the Turing machine (1936).

Also in 1936, Alonzo Church invented the lambda calculus, a simple (but very different) model for computation based on functions:

\[ t ::= x \mid (\lambda x. t) \mid (t \ t) \]

John McCarthy (LISP, 1958) and others recognized that this allows for a more declarative programming style, which focuses on what programs should accomplish, rather than describing how to go about it.
We know from Euclid that:

\[
\begin{align*}
gcd(0, n) &= n & \text{if } n > 0 \\
gcd(m, n) &= gcd(n \mod m, m) & \text{if } m > 0
\end{align*}
\]
/* PRE: m,n >= 0 and m+n > 0
 * POST: the greatest common divisor of m and n
 */
int gcd(int m, int n) {
    int a=m, b=n, prevA;
    /* INVARIANT: gcd(m,n) = gcd(a,b) */
    while (a != 0) {
        prevA = a;
        a = b % a;
        b = prevA;
    }
    return b;
}
GCD in a Functional Language (SML)

\[
\begin{align*}
(* & \text{ PRE: } m, n \geq 0 \text{ and } m+n > 0 \\
& \text{ POST: the greatest common divisor of } m \text{ and } n *) \\
\text{fun} & \quad \text{gcd} \ (0, \ n) = n \\
& \quad \text{gcd} \ (m, \ n) = \text{gcd} \ (n \ \text{mod} \ m, \ m)
\end{align*}
\]
Features of Imperative vs. Functional Programs

<table>
<thead>
<tr>
<th>Imperative</th>
<th>Functional</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sequence of instructions</td>
<td>Evaluation of expressions</td>
</tr>
<tr>
<td>Side effects (e.g., memory updates)</td>
<td>Side-effect free</td>
</tr>
<tr>
<td>Loops</td>
<td>Recursion</td>
</tr>
<tr>
<td>Functions operate on data</td>
<td>Functions are first-class</td>
</tr>
<tr>
<td>Describe <em>how</em> to compute a value</td>
<td>Describe <em>what</em> to compute</td>
</tr>
<tr>
<td>Close to the hardware</td>
<td>Close to the actual problem</td>
</tr>
</tbody>
</table>
When to Use Functional Programming?

Functional programming languages are Turing-complete: they can compute exactly the same functions as imperative and other languages.

Always? ↔ Never?

How do you choose a programming language, anyway?
Strengths of Functional Programming

- **Unit testing**: without global state, a function’s return value only depends on its arguments. Each function can be tested in isolation.
- **Concurrency**: without modifiable data, programs can easily be parallelized (cf. Google’s MapReduce). There is no risk of data races.
- **Correctness**: functional programs are much easier to reason about than programs that manipulate state.

Further reading: [http://www.defmacro.org/ramblings/fp.html](http://www.defmacro.org/ramblings/fp.html)
Recursion
Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem.

http://en.wikipedia.org/wiki/Recursion_%28computer_science%29
Recursion: Examples

Summing over a list:

```hs
fun sum [ ] = 0
    | sum (x :: xs) = x + sum xs
```

Factorial function:

```hs
fun fac 0 = 1
    | fac n = n * fac (n-1)
```
Recursion is one of the key concepts in functional programming.

You will use (some form of) recursion wherever you might use a loop in an imperative language.
Factorial Revisited

\[
\text{fun } \text{fac } 0 = 1 \\
| \quad \text{fac } n = n \times \text{fac } (n-1)
\]
Factorial Revisited

```
fun fac 0 = 1
  | fac n = n * fac (n-1)
```

What happens if \( n < 0 \)?
We test if the pre-condition $n \geq 0$ is satisfied (defensive programming).

```
fun fac n =
  if n < 0 then raise Domain
  else if n = 0 then 1
  else n * fac (n-1)
```
We test if the pre-condition $n \geq 0$ is satisfied (defensive programming).

```plaintext
fun fac n =
  if n < 0 then raise Domain
  else if n = 0 then 1
  else n * fac (n-1)
```

Useless test of the pre-condition at each recursive call.
Factorial Revisited (cont.)

We introduce an auxiliary function.

```ml
fun fac n =
  let
    fun fac' 0 = 1
    | fac' n = n * fac' (n-1)
  in
    if n < 0 then raise Domain
    else fac' n
  end
```

In `fac`: pre-condition verification
In `fac'`: no pre-condition verification
Exponentiation: Specification and Construction

Specification:

fun expo x n

TYPE: real -> int -> real

PRE: n ≥ 0

POST: x^n
Exponentiation: Specification and Construction

Specification:

```fun expo x n
```

**TYPE:** real → int → real

**PRE:** $n \geq 0$

**POST:** $x^n$

Construction:

Error case: $n < 0$: raise an exception

Base case: $n = 0$: result is 1

Recursive case: $n > 0$: result is $x^n = x \times x^{n-1} = x \times \text{expo} \ x \ (n - 1)$
Exponentiation: Program

\[
\text{fun } \text{expo} \ x \ n = \\
\begin{array}{l}
\text{let } \\
\quad \text{fun } \text{expo}' \ x \ 0 = 1 \\
\quad \quad | \ \text{expo}' \ x \ n = x \ast \text{expo}' \ x \ (n-1) \\
\end{array} \\
\text{in } \\
\begin{array}{l}
\quad \text{if } n < 0 \text{ then } \\
\qquad \text{raise Domain} \\
\quad \text{else } \\
\qquad \text{expo}' \ x \ n \\
\end{array} \\
\text{end}
\]

Observation: the first argument of expo' never changes; it is always x. Let’s get rid of it.
Exponentiation: Program

fun expo x n =
  let
    fun expo' x 0 = 1
    | expo' x n = x * expo' x (n-1)
in
  if n < 0 then raise Domain
  else
    expo' x n
end

Observation: the first argument of expo’ never changes; it is always x. Let’s get rid of it.
fun expo x n =
  let
    fun expo' 0 = 1
    | expo' n = x * expo' (n-1)
  in
    if n < 0 then raise Domain
    else
      expo' n
  end
Triangle: Specification and Construction

Specification:

fun triangle a b

TYPE: int → int → int

PRE: true

POST: \( \sum_{i=a}^{b} i \)
Triangle: Specification and Construction

Specification:

fun triangle a b

TYPE: int -> int -> int

PRE: true

POST: \[ \sum_{i=a}^{b} i \]

Construction:

Error case: (none)

Base case: \( a > b \): result is 0

Recursive case: \( a \leq b \): result is

\[ \sum_{i=a}^{b} i = a + (\sum_{i=a+1}^{b} i) = a + \text{triangle} \ (a + 1) \ b \]
fun triangle a b =
  if a > b then
    0
  else
    a + triangle (a+1) b
Recursion: Correctness

How do we know what a recursive program computes?

Example:

```latex
fun f 0 = 0
  | f n = 1 + f (n-1)
```

What does $f$ compute?
How do we know what a recursive program computes?

Example:

```haskell
fun f 0 = 0
| f n = 1 + f (n - 1)
```

What does \( f \) compute?

Answer: \( f(n) = n \), if \( n \geq 0 \)
Seems reasonable, but how do we prove it?
The Axiom of Induction

If $P$ is a property of natural numbers such that

1. $P(0)$ is true, and
2. whenever $P(k)$ is true, then $P(k + 1)$ is true,

then $P(n)$ is true for all natural numbers $n$. 
Example: Proof by Induction

```haskell
fun f 0 = 0
  | f n = 1 + f (n-1)
```

We want to prove that $f(n) = n$ for all natural numbers $n$. (So, in this example, $P(n) \equiv f(n) = n$.)

1. Base case $P(0)$: $f(0) = 0$ by definition.
2. Inductive step: assume that $P(k)$ is true, i.e., $f(k) = k$. Then

\[
f(k + 1) = 1 + f((k + 1) - 1) = 1 + f(k) = 1 + k = k + 1
\]

hence $P(k + 1)$ is true.

It follows that $f(n) = n$ for all natural numbers $n$. 
Another Example: Proof by Induction

\[
\text{fun } g \ 0 = 0 \\
\quad | \ g \ n = n + g \ (n-1)
\]

What does \( g \) compute?
Another Example: Proof by Induction

\begin{verbatim}
fun g 0 = 0
| g n = n + g (n-1)
\end{verbatim}

What does \( g \) compute?

Answer: \( g(n) = \frac{n(n+1)}{2} \)

Proof (by induction): exercise.
Complete Induction

**Complete induction** is a variant of induction that allows to assume the induction hypothesis not just for the immediate predecessor, but for all smaller natural numbers.

If \( P \) is a property of natural numbers such that

1. \( P(0) \) is true, and
2. whenever \( P(j) \) is true for all \( j \leq k \), then \( P(k + 1) \) is true,

then \( P(n) \) is true for *all* natural numbers \( n \).

Exercise: show that complete induction is equivalent to the axiom of induction.
Correctness of Functional Programs

Suppose we want to show that a recursive function

```haskell
fun f x = ... f ( ... ) ... 
```

satisfies some property \( P(x, f(x)) \).

Solution:

1. Show that \( f \) terminates (for all values of \( x \) that we care about).
2. Assume that all recursive calls \( f(x') \) satisfy the property \( P(x', f(x')) \), and show \( P(x, f(x)) \).

(This is just an induction proof in disguise.)
Example: Correctness of Functional Programs

```
fun fac 0 = 1
  | fac n = n * fac (n-1)
```

We want to show that \( P(n, \text{fac}(n)) \equiv \text{fac}(n) = 1 \ast \cdots \ast n \) holds.

1. For now, let’s just assume that \( \text{fac} \) terminates for all \( n \geq 0 \). (We’ll actually prove this in a few minutes.)

2. Assume that the recursive call satisfies \( P(n - 1, \text{fac}(n - 1)) \equiv \text{fac}(n - 1) = 1 \ast \cdots \ast (n - 1) \). Now show \( P(n, \text{fac}(n)) \):

   (i) If \( n = 0 \), \( \text{fac}(0) = 1 \) as required.
   (ii) If \( n > 0 \), \( \text{fac}(n) = n \ast \text{fac}(n - 1) = n \ast (1 \ast \cdots \ast (n - 1)) = 1 \ast \cdots \ast n \) using algebraic properties of multiplication.
Construction Methodology

Objective: construction of a (recursive) SML program computing the function $f : D \rightarrow R$ given a specification $S$.

Methodology:

1. **Case analysis**: identify error, base, and recursive case(s)

2. **Partial correctness**: show that the base case returns the correct result; show that the recursive cases return the correct result, *assuming* that all recursive calls do.

3. **Termination**: find a suitable variant.
A **variant** for a (recursive) function is any expression over the function’s parameters that takes values in some ordered set $A$ such that

- there are no infinite descending chains $v_0 > v_1 > \ldots$ in $A$, and
- for any recursive call, the variant decreases strictly.

Often, $A = \{0, 1, 2, \ldots\}$.

Variants are often simple: e.g., a non-negative integer given by a parameter or the size of some input data. But watch out for the more difficult cases!
Example: Variants

\[
\text{fun } \text{fac } 0 = 1 \\
| \text{fac } n = n \times \text{fac } (n-1)
\]

Variant for fac n:
**Example: Variants**

```haskell
fun fac 0 = 1
| fac n = n * fac (n-1)
```

Variant for `fac n`: `n`

This variant is a non-negative integer (thus, there are no infinite descending chains) that strictly decreases with every recursive call (`n - 1 < n`).
Forms of Recursion

So far: **simple** recursion (one recursive call only, some variant is decremented by one) — corresponds to simple induction

Other forms of recursion:

- Complete recursion
- Multiple recursion
- Mutual recursion
- Nested recursion
- Recursion on a generalized problem
Example: Complete Recursion

Specification:

**fun** int\_div \ a \ b

**TYPE:** int \ −\> \> int \> int \> int

**PRE:** \ a \ ≥ \> 0 \ and \ b \> 0

**POST:** \ (q, r) \ such \ that \ a = q \> b \> r \> and \ 0 \≤ \> r \< b
Example: Complete Recursion

Specification:

```fun`` int\_div a b

TYPE: int \(\rightarrow\) int \(\rightarrow\) int * int

PRE: \(a \geq 0\) and \(b > 0\)

POST: \((q, r)\) such that \(a = q \cdot b + r\) and \(0 \leq r < b\)

Construction:

Error case: \(a < 0\) or \(b \leq 0\): raise an exception

Base case: \(a < b\): since \(a = 0 \cdot b + a\), result is \((0, a)\)

Recursive case: \(a \geq b\): since \(a = q \cdot b + r\) iff \(a - b = (q - 1) \cdot b + r\), \(\text{int\_div } (a-b) b\) will compute \(q - 1\) and \(r\)
fun int_div a b =
  let
    fun int_div' a b =
      if a < b then
        (0, a)
      else
        let
          val (q, r) = int_div' (a-b) b
        in
          (q+1, r)
        end
      end
  in
    if a < 0 orelse b <= 0 then
      raise Domain
    else
      int_div' a b
  end

To prove correctness of int_div, we need an induction hypothesis not only for a – 1, but for all values less than a, i.e., we need complete induction.
Example: Multiple Recursion

Definition of the Fibonacci numbers:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n - 1) + \text{fib}(n - 2)
\end{align*}
\]
Example: Multiple Recursion

Definition of the Fibonacci numbers:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n - 1) + \text{fib}(n - 2)
\end{align*}
\]

Specification:

```fun fib n
```

TYPE: int \(\rightarrow\) int

PRE: \(n \geq 0\)

POST: fib\((n)\)
Example: Multiple Recursion

Definition of the Fibonacci numbers:

\[
\begin{align*}
\text{fib}(0) &= 0 \\
\text{fib}(1) &= 1 \\
\text{fib}(n) &= \text{fib}(n-1) + \text{fib}(n-2)
\end{align*}
\]

Specification:

\textbf{fun} \text{fib} \ n

\textbf{TYPE:} int → int

\textbf{PRE:} n ≥ 0

\textbf{POST:} fib(n)

Program:

\textbf{fun} \text{fib} \ n =

\textbf{let}

\textbf{fun} \text{fib'} 0 = 0

| \text{fib'} 1 = 1

| \text{fib'} n = \text{fib'} (n-1) + \text{fib'} (n-2)

\textbf{in}

\textbf{if} n < 0 \textbf{then raise Domain}

\textbf{else} \text{fib'} n

\textbf{end}
Example: Mutual Recursion

Recognizing even and odd natural numbers:

fun even n
TYPE: int → bool
PRE: \( n \geq 0 \)
POST: true iff \( n \) is even

fun odd n
TYPE: int → bool
PRE: \( n \geq 0 \)
POST: true iff \( n \) is odd

Program:

fun even 0 = true
| even n = odd (n−1)
and odd 0 = false
| odd n = even (n−1)

Mutual recursion requires simultaneous declaration of the functions and global correctness reasoning.
Example: Nested Recursion

The Ackermann function:

```plaintext
fun acker 0 m = m + 1
| acker n 0 = acker (n-1) 1
| acker n m = acker (n-1) (acker n (m-1))
```

Variant?
The Ackermann function:

```ml
fun acker 0 m = m + 1
| acker n 0 = acker (n - 1) 1
| acker n m = acker (n - 1) (acker n (m - 1))
```

Variant? The pair \((n, m)\) ∈ \(\mathbb{N} \times \mathbb{N}\), where \(\mathbb{N} \times \mathbb{N}\) is ordered lexicographically: \((a, b) \preceq_{\text{lex}} (c, d)\) iff \(a < c\) or \((a = c\) and \(b < d\)).
Example: recognizing prime numbers

Specification:

```plaintext
fun prime n
TYPE: int -> bool
PRE: n > 0
POST: true iff n is a prime number
```
Recursion on a Generalized Problem

Example: recognizing prime numbers

Specification:

fun prime n
TYPE: int -> bool
PRE: n > 0
POST: true iff n is a prime number

Construction:

It seems impossible to directly determine whether n is prime if we only know whether \( n - 1 \) is prime. We thus need to find a function

- that is more general than prime, in the sense that prime is a special case of this function, and
- for which a recursive program can be constructed.
Specification of the generalized function:

```haskell
fun indivisible n low up
TYPE: int -> int -> int -> bool
PRE: n, low, up ≥ 1
POST: true iff n has no divisor in \{low, \ldots, up\}
```
Recursion on a Generalized Problem (cont.)

Specification of the generalized function:

fun indivisible n low up
TYPE: int -> int -> int -> bool
PRE: n, low, up ≥ 1
POST: true iff n has no divisor in \{low, \ldots, up\}

Construction:

Base case: low > up: result is true
Recursive case: low ≤ up: n has no divisor in \{low, \ldots, up\} iff low does not divide n and n has no divisor in \{low + 1, \ldots, up\}
Recursion on a Generalized Problem (cont.)

Program:

```haskell
fun indivisible n low up =
  low > up otherwise
  (n mod low <> 0 andalso indivisible n (low + 1) up)
```

Now the function `prime` is essentially a special case of `indivisible`:

```haskell
fun prime n = if n <= 0 then raise Domain else n > 1 andalso indivisible n (n - 1)
```
Program:

```
fun indivisible n low up =
  low > up orelse
  (n mod low <> 0 andalso indivisible n (low + 1) up)
```

Now the function prime is essentially a special case of indivisible:

```
fun prime n =
  if n <= 0 then
    raise Domain
  else
    n > 1 andalso indivisible n 2 (n-1)
```
Standard Methods of Generalization

- Let the recursive function take additional parameters, so that the problem we want to solve is a special case.
- Let the recursive function return more information than is required in the problem statement.

Exercise: implement a function that computes fib(n) with a number of recursive calls proportional to \(n\).
Let the recursive function take additional parameters, so that the problem we want to solve is a special case.

Let the recursive function return more information than is required in the problem statement.

Exercise: implement a function that computes \( \text{fib}(n) \) with a number of recursive calls proportional to \( n \).